

# Evolving difficult SAT instances thanks to local search

Olivier Bailleux, Université de Bourgogne

November 29, 2010

## Abstract

We propose to use local search algorithms to produce SAT instances which are harder to solve than randomly generated  $k$ -CNF formulae. The first results, obtained with rudimentary search algorithms, show that the approach deserves further study. It could be used as a test of robustness for SAT solvers, and could help to investigate how branching heuristics, learning strategies, and other aspects of solvers impact their robustness.

## 1 Introduction

Because the hardest instances for a given solver are not necessarily the hardest ones for another solver, it is difficult to evaluate and to compare the performances of SAT solvers. In the competitions like [5], the solvers are evaluated with several kinds of instances (random, hand crafted, industrial...), and the results show that no solver outperforms the other ones in all the categories.

The underlying question is about the *robustness* of SAT solvers. How to identify the hardest instances for a given solver ? We propose to use local search algorithms to find hard instances.

## 2 Example

In this section, we report the results of a first prospective experimentation using the SAT4J framework [2]. The SAT solver provided by the JAVA library SAT4J can either be used as a stand alone solver or integrated into a `java` application. We used this second way to develop a prototype allowing the evolution of a SAT instance by local search.

A SAT instance is a CNF propositional formula, i.e., a conjunction of clauses, where each clause is a disjunction of literals. Each literal is either a propositional variable or a negated propositional variable. Our initial formula contains  $m$  randomly generated clauses of  $k$  literals. The variables are uniformly drawn in a set of  $n$  variables under the constraint that no variable can occur more than one times in any clause. Each variable is negated with probability  $1/2$ .

We propose two ways to evaluate the hardness of an instance, which are the number of propagations and the number of decisions required to solve it. A propagation consists to fix a variable thanks to the unit resolution rule, which is the basic filtering method used in SAT solvers. A decision is a binary node of the search tree.

The used local search process aims to increase the number  $p$  of propagations the solver needs to solve the current formula. At each generation, the current instance is modified by replacing an existing randomly drawn clause by a new randomly generated clause. Then the solver is ran and the new number  $q$  of propagations is compared to  $p$ . If  $q < p$  then the old current formula is restored, else the new one is kept.

Figure 1 show the evolution of the number of propagations and the number of decisions during  $10^7$  generations. The required number of propagations increases two orders of magnitude, while the required

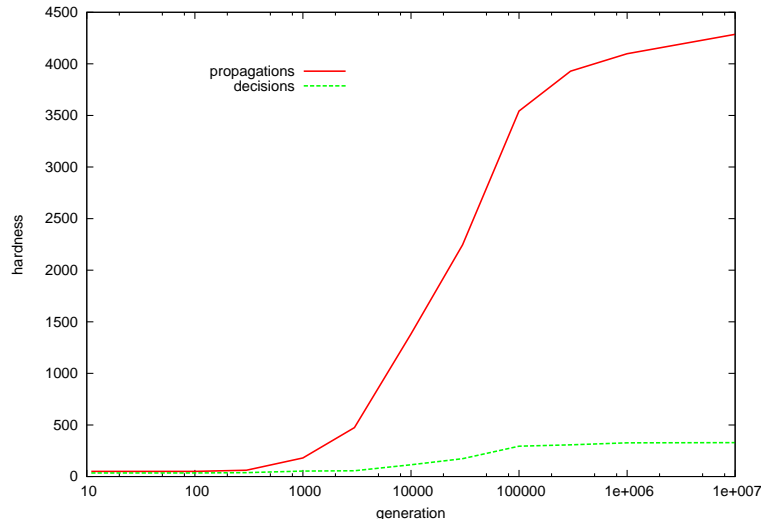


Figure 1: Evolving a CNF formula with 100 clauses and 50 variables

number of decisions increases one order of magnitude. The instance remains satisfiable throughout the evolution.

### 3 Evolutions strategies

The local search algorithm used in section 2 is based on a rudimentary greedy search strategy with a transition operator that replaces a randomly drawn clause by a new randomly generated one. Two ways can be explored toward more sophisticated evolution schemes.

#### 3.1 Search strategies

The aim of a local search strategy is essentially to avoid the stance of evolution, which may be due to local extremums. We propose three research directions.

1. The algorithm presented in section 2 could be improved by modifying several clauses simultaneously when the score (i.e., the number of propagations, the number of decisions, or any other relevant indicator) does not increase for some time. Such a "break" is expected to allow the search process to escape from the current basin of attraction (if applicable), with the hope of finding a more promising evolution path.
2. Another idea is to try guiding the search by focusing it on some variables : A weight is assigned to each variable. When changing a clause improve the current score, the weights of the related variables increases. The clauses containing variables with highest weight are preferentially modified. On the contrary, when changing a clause does not improve the current score, the weights of the corresponding variables decreases.
3. Population based search techniques, like genetic algorithms, could be used. This suppose to find relevant crossover operators, in the sense that the offspring of two (or more) formulae should be likely to be as hard as its parents. An idea worth exploring is first to aggregate two formulae,

then reduce the size of the resulting offspring tanks to a local search process designed to remove clauses. A more standard crossover operator is used in [6] with convincing results.

Moreover, it is also possible to start the search with a formula already known to be hard with respect to its size.

### 3.2 Transition operators

It is known that the choice of the transition (or neighborhood) operator can dramatically impact the efficiency of a local search process. In section 1, this operator consists to replace a clause by a new one. The number of clauses remains constant during the search. Other ways can be considered, like the following ones.

1. Instead of changing a whole clause, we can try to change one literal at each generation.
2. Sometimes, a new clause can be added, or an existing one can be removed.
3. The current instance can be maintained satisfiable or unsatisfiable by rejecting the candidates that do not verify the criterion.

## 4 A few experiments

In this section, we present the first results obtained with a local search algorithm designed to evolve an unsatisfiable formula in two stages. The initial formula is randomly generated with a ratio number of clauses / number of variables which is substantially above the satisfiability threshold [3], in such a way that the obtained SAT instance is most likely unsatisfiable. In the first stage, the transition operator consists in removing a clause and the selection criteria accepts the new formula if it remains unsatisfiable. The duration of this stage is 10 times the number of clauses. In the second stage, the transition operator consists in replacing a randomly chosen clause by a new randomly generated one. The selection criterion accepts the new formula if the number of decisions required to solve it does not decrease.

Figure 2 shows the evolution of both the size of the 3-CNF formula of 50 variables and the number of decision required to solve it on  $10^5$  generations. The second stage begins after about 1000 generations. Figure 3 shows the evolution of a 3-CNF formula of 100 variables in the same conditions. The number of decisions increases from 122 to 1572 during the first stage, where the number of clauses decreases from 600 to 280. In the second stage, the number of clauses remains constant while the numbers of decisions increase up to 2180. We examined whether the hardness of this instance also increases with respect to another SAT solver, i.e., cryptominisat [4] : the initial formula requires 91 decisions, and the final one 2051 decisions.

For a first glimpse of how the initial instance impacts the result of the evolution, Figure 4 shows the initial and final number of decisions for twenty formula of 50 variables : there is no apparent correlation between the initial and final hardness, which is consistently greatly increased.

## 5 Related works

We presented the idea of using local search for finding hard SAT instances in [1], but this work, although promising, has not been resumed since.

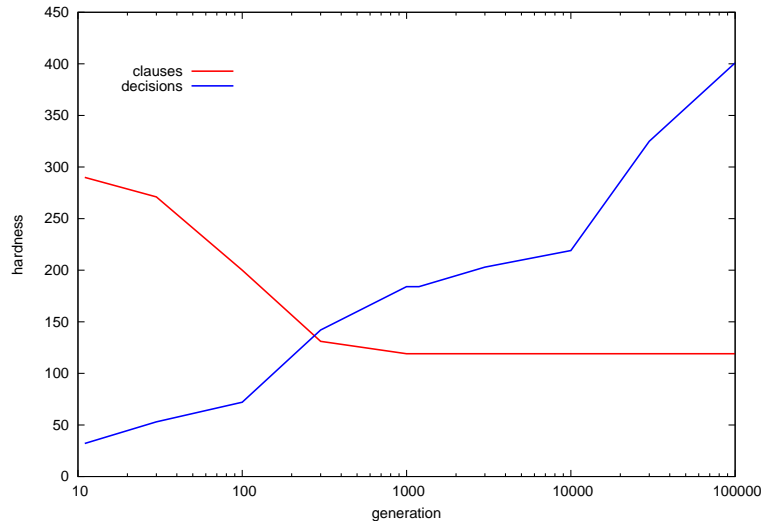


Figure 2: Evolving an unsatisfiable CNF formula with 50 clauses and 300 variables

To the best of our knowledge, the only other work really close to our approach is [6], which use genetic algorithms to evolve combinatorial problem instances in order that there become difficult to solve. This already fairly advanced work proposes an analysis of the generated instances for identifying what make them hard to solve.

## 6 Concluding remarks

We proposed to use local search algorithms to produce SAT instances which are harder to solve than randomly generated  $k$ -CNF formulae.

The specificity of the proposed approach is that the computation of the objective function is very expensive, because the aim is to produce hard instances, with the result to increase the cost of evaluate the hardness of these instances.

Nevertheless, the first results obtained with rudimentary search algorithms show that the approach deserves further study. It could be used as a test of robustness for SAT solvers, and could help to investigate how branching heuristics, learning strategies, and other criteria, impact this robustness.

## References

- [1] Olivier Bailleux and Jacqueline Chabrier. Local search against local search. ECAI’96 Workshop on Advances in Propositional Deduction, August 1996.
- [2] Daniel Le Berre and Anne Parrain. The sat4j library, release 2.2. *JSAT*, 7:59–64, 2010.
- [3] Olivier Dubois. Upper bounds on the satisfiability threshold. *Theoretical Computer Science*, 265(1-2):187 – 197, 2001.
- [4] M. Soos, K. Nohl, and C. Castelluccia. Extending sat solvers to cryptographic problems. In *proc. of SAT 09*, LNCS 5584, pages 244–257. Springer, 2009.

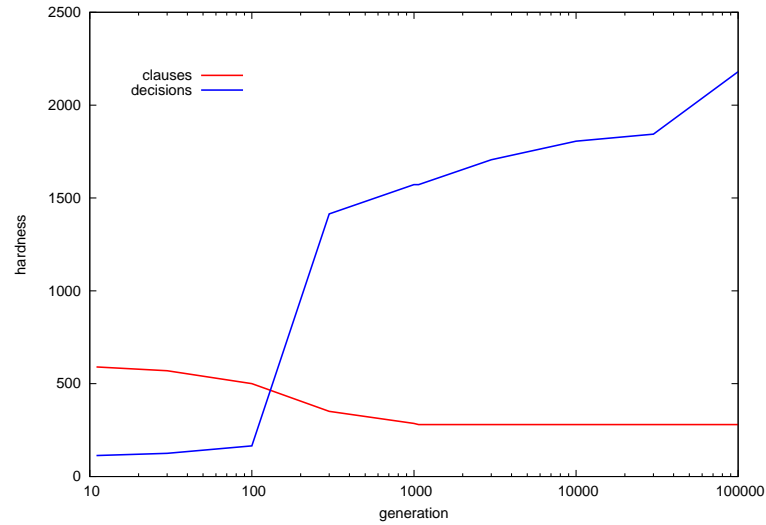


Figure 3: Evolving an unsatisfiable CNF formula with 100 clauses and 600 variables

- [5] Ewald Speckenmeyer, Chu Min Li, Vasco Manquinho, and Armando Tacchella. *JSAT Special Issue on the 2007 Competitions*. IOS Press, 2007.
- [6] Jano I. van Hemert. Evolving combinatorial problem instances that are difficult to solve. *Evolutionary Computation*, 14(4), 2006.

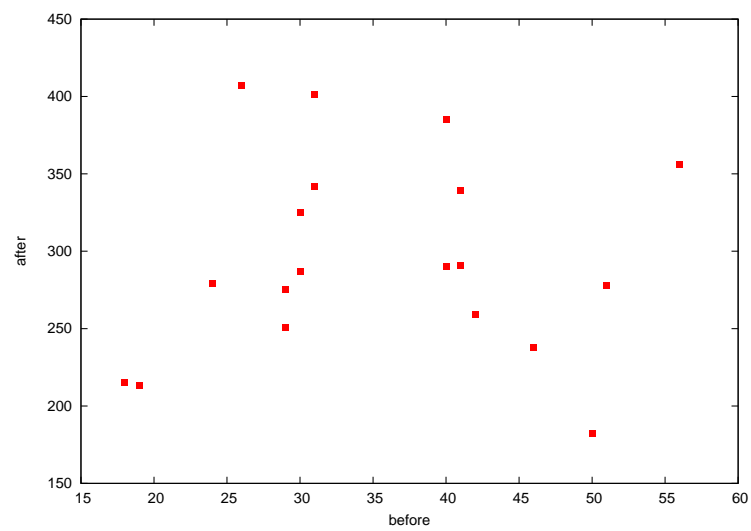


Figure 4: Evolving 20 unsatisfiable CNF formula with 50 clauses and 300 variables, initial versus final score